下载APP

25 | GORM：数据库的使用必不可少（上）

2021-11-12 叶剑峰

《手把手带你写一个Web框架》 课程介绍 >

讲述：叶剑峰
时长 20:36 大小 18.87M

你好，我是轩脉刃。

一个 Web 应用，有很大部分功能是对数据库中数据的获取和加工。比如一个用户管理系统，我们在业务代码中需要频繁增加用户、删除用户、修改用户等，而用户的数据都存放在数据库中。所以对数据库的增删改查，是做 Web 应用必须实现的功能。而我们的 hade 框架如何更好地支持数据库操作呢？这两节课我们就要讨论这个内容。

ORM

提到数据库，就不得不提 ORM 了，有的同学一接触 Web 开发，就上手使用 ORM 了，这里我们要明确一点：ORM 并不等同于数据库操作。

数据库操作，本质上是使用 SQL 语句对数据库发送命令来操作数据。而 ORM 是一种将数据库中的数据映射到代码中对象的技术，这个技术的需求出发点就是，**代码中有类，数据库中有数据表，我们可以将类和数据表进行映射，从而使得在代码中操作类就等同于操作数据库中的数据表了**。

ORM 这个概念出现的时间无从考究了，基本上从面向对象的编程思想出来的时候就有讨论了。但是到现在，是否要使用 ORM 的讨论也一直没有停止。

不支持使用 ORM 的阵营的观点基本上是使用 ORM 会影响性能，且会让使用者不了解底层的具体最终拼接出来的 SQL，容易造成用不上索引或者最终拼接错误的情况。而支持使用 ORM 的阵营的观点主要是它能切切实实加速应用开发。

就我个人的观点和经验，我还是支持使用 ORM 的。我认为 ORM 不仅仅是一种映射技术，也是一种建模思想。**因为数据库是和业务紧密关联起来的，建立数据库表结构的时候，也是建立了一个业务模型**。使用代码中的类定义，比如定义了一个 User 类，基本上就定义了一个 User 表，这样也是一个建立业务模型的过程。

其实不论 ORM 的讨论如何激烈，基本上各个语言都已经有了 ORM 的实现，比如 Java 的 Hibernate、PHP 的 Doctrine、Ruby 的 ActiveRecord。而在 Golang 中，现在最流行的 ORM 库是国人的开源项目 Gorm 。

Gorm 作者 Jinzhu 目前是字节跳动的员工，他在 GitHub 上开源共享了诸如 copier、configer 等开源项目，Gorm 目前 star 数有 26k 之多，使用 MIT 的许可证协议，项目启动于 2013 年，目前是 v2 版本。

这个 **v2 版本对应 Gorm GitHub 上 v1.20 以上的 tag**，这点我们要额外注意。因为网上的分析文章很多都是基于 Gorm 的 v1 版本写的，但 Gorm 的 v1 和 v2 版本相差比较大。所以在看 Gorm 文章的时候需要先明确下是什么版本。

我们的框架侧重于整合，站在巨人的肩膀上才更符合现代化框架的要求。基于此， hade 框架并不打算重新开发一套 ORM 框架，而是会直接融合 Gorm 框架成为我们容器中的一个服务 orm service。

版本选择的是 Gorm 截止 2021/10/23 日最新的 v1.21.16 的 tag。毕竟 Gorm 是个有一定体量的项目，而且理解它的重点部分源码的实现原理，对使用者来说非常重要，值得我们先花一章来学习理解。如何融合 Gorm，我们下节课继续学。

Gorm

一个 ORM 库，最核心要了解两个部分。一个部分是数据库连接，它是怎么和数据库建立连接的，第二部分是数据库操作，即它是怎么操作数据库的。

我们看一个最精简的 Gorm 的使用例子：

复制代码

```
package main

import (
  "gorm.io/gorm"
  "gorm.io/driver/mysql"
)

// 定义一个 gorm 类
type User struct {
    ID           uint
    Name         string
}

func main() {
  // 创建 mysql 连接
  dsn := "xxxxxxx"
  db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
  ...

  // 插入一条数据
  db.Create(&User{Name: "jianfengye"})

  ...
}
```

main 函数，先创建一个 MySQL 连接，再插入一条数据，这个 User 数据是通过事先定义好的 User 结构来进行设置的。**其中的 gorm.Open 就是一个快速连接数据库的接口，而后续的 Create 是如何操作数据库的接口。**

我们今天的任务就是理解这几行代码的实现原理，后面会不断拿这个例子举例。

数据结构

先把重点放在理解这个 Open 函数上，因为这个函数包含了 Gorm 中关键的几个对象，把这些关键数据结构一一理解透，再跟踪具体的源码能事半功倍。另外也推荐你边看边自己画出这几个关键参数的关系，非常有助于理解和记忆，每个参数讲完之后我也会展示一下我画的分析图供你参考。

来看它的源码定义：

复制代码

```
// 初始化数据库连接
func Open(dialector Dialector, opts ...Option) (db *DB, err error)
```

这个初始化数据库链接的 Open 函数有两个参数 dialector、opts，和两个返回值 gorm.DB、error。我们先理解下这几个参数的意义。

Dialector

第一个参数是 Dialector，这是什么呢？它代表数据库连接器。这里也是一个面向接口编程的思想，连接器结构 Dialector 是一个接口，代表如果你要使用 Gorm 来连接你的数据库，那么，只需要实现这个接口定义的所有方法，就可以使用 Gorm 来操作你的数据库了。

所以，这个接口 Dialecotor 中定义的所有方法，都是在后续的查询、更新、数据库迁移等操作中会使用到的。具体每个方法在哪里使用到的，如果你感兴趣可以跟踪下去，如果你不感兴趣也无所谓，只需要记得在后续某个 gorm 接口的具体实现中会用到就行。

复制代码

```
// Dialector GORM database dialector
type Dialector interface {
    Name() string // 连接器名称
    Initialize(*DB) error // 连接器初始化连接方法
    Migrator(db *DB) Migrator // 数据库迁移方法
    DataTypeOf(*schema.Field) string // 类中每个字段的类型对应到 sql 语句
    DefaultValueOf(*schema.Field) clause.Expression // 每个字段的默认值对应到 sql i
    BindVarTo(writer clause.Writer, stmt *Statement, v interface{}) // 使用预编译
    QuoteTo(clause.Writer, string) // 将类中的注释对应到 sql 语句中
    Explain(sql string, vars ...interface{}) string // 将有占位符的 sql 解析为无占位
}
```

不同的数据库有不同的 Dialector 实现，我们称之为“驱动”。每个数据库的驱动，都有一个 git 地址进行存放。目前 gorm 官方支持五种数据库驱动：

MySQL 的 Gorm 驱动地址为 gorm.io/driver/mysql

Postgres 的 Gorm 驱动地址为 gorm.io/driver/postgres

SQLite 的 gorm 驱动地址为 gorm.io/driver/sqlite

SQL Server 的 gorm 驱动地址为 gorm.io/driver/sqlserver

ClickHouse 的 gorm 驱动地址为 gorm.io/driver/clickhouse

如果要创建对应数据库的连接，要先引入对应的驱动。而在对应的驱动库中都有一个约定的 Open 方法，来创建一个新的数据库驱动。比如要创建 MySQL 的连接，使用下面这个例子：

复制代码

```
import (
  "gorm.io/driver/mysql"
  "gorm.io/gorm"
)

// 创建连接
dsn := "gorm:gorm@tcp(localhost:9910)/gorm?charset=utf8&parseTime=True&loc=Loc
db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
```

我们看到这里有个 mysql.Open，就是创建 MySQL 的 Gorm 驱动用的。而这个 Open 函数只有一个字符串参数 DSN，这个参数可能有的同学还不是很了解，我们一起研究下。

DSN

DSN 全称叫 Data Source Name，数据库的源名称。

DSN 定义了一个数据库的连接方式及信息，包含用户名、密码、数据库 IP、数据库端口、数据库字符集、数据库时区等信息。可以说**一个 DSN 就是一个数据源的描述**。但是 DSN 并没有明确的官方文档要求其格式，每个语言、每个平台都可以自己定义 DSN 格式，只要定义和解析能对得上就行。

在社区中，大家普遍会按照以下这种格式来进行定义：

```
scheme://username:password@host:port/dbname?param1=value1&param2=value2&...
```

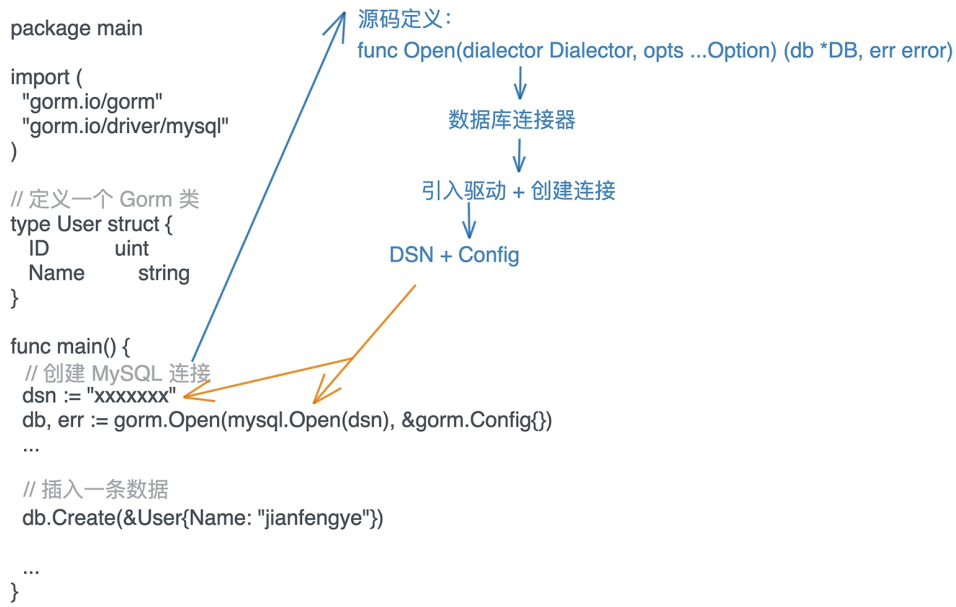
比如通过 Unix 的 socket 句柄连接本机 MySQL：

```
mysql://user@unix(/path/to/socket)/dbname
```

通过 TCP 连接远端 postgres：

```
pgsql://user:pass@tcp(localhost:5555)/dbname
```

DSN 在 gorm 中的使用就如下图所示，我们使用这个 dsn 结合具体的驱动来生成 Open 函数的第一个参数，数据库连接器。



极客时间

在具体使用中，我们当然可以直接执行字符串拼接，来拼接出一个 DSN，但是我们更希望能**通过定义一个 Golang 的数据结构自动拼接出一个 DSN，或者是从一个 DSN 字符串反序列化生成这个数据结构。**

在 Golang 中有一个第三方库 github.com/go-sql-driver/mysql 就提供了这样的功能。这个库用来对 Go 中的 SQL 提供 MySQL 驱动，其中定义了一个 Config 结构，能映射到 DSN 字符串。Config 结构中一些比较重要的字段说明，我写在注释中了：

复制代码

```
type Config struct {
    User              string            // 用户名
    Passwd            string            // 密码 (requires User)
    Net               string            // 网络类型
    Addr              string            // 地址 (requires Net)
    DBName            string            // 数据库名
    Params            map[string]string // 其他连接参数
    Collation         string            // 字符集
    Loc               *time.Location    // 时区
    MaxAllowedPacket  int               // 最大包大小
    ServerPubKey      string            // 连接公钥名称
    pubKey            *rsa.PublicKey    // 连接公钥 key
    TLSConfig         string            // TLS 的配置名称
    tls               *tls.Config       // TLS 的配置项
    Timeout           time.Duration     // 连接超时
    ReadTimeout       time.Duration     // 读超时
    WriteTimeout      time.Duration     // 写超时

    ...
    CheckConnLiveness       bool // 在使用连接前确认连接可用
    ...
    ParseTime               bool // 是否解析时间格式
    ...
}
```

从 DSN 到这个 Config 结构，我们使用 github.com/go-sql-driver/mysql 的 ParseDSN ，而从 Config 结构到 DSN 我们使用 FormatDSN 方法：

复制代码

```
// 解析 dsn
func ParseDSN(dsn string) (cfg *Config, err error)

// 生成 dsn
func (cfg *Config) FormatDSN() string
```

这两个方法都先记下，下节课会用到。

Option

第一个初始化数据库的参数 dialector 以及之前必要的驱动引入相关参数 DSN，就讲解到这里。我们回头继续看 Open 函数：

复制代码

```
// 初始化数据库连接
func Open(dialector Dialector, opts ...Option) (db *DB, err error)
```

第二个参数 opts 是 Option 的可变参数，而这个 Option 是一个实现了 Apply 和 AfterInitialize 的接口：

复制代码

```
// Option 接口
type Option interface {
    Apply(*Config) error
    AfterInitialize(*DB) error
}
```

这种可变参数如何使用呢？我们看下 Open 的源码：

复制代码

```
// Open 初始化 DB 的 Session
func Open(dialector Dialector, opts ...Option) (db *DB, err error) {
    config := &Config{}

    ...

    for _, opt := range opts {
        if opt != nil {
            // 先调用 Apply 初始化 Config
            if err := opt.Apply(config); err != nil {
                return nil, err
            }
            // Open 最后结束后调用 AfterInitialize
            defer func(opt Option) {
                if errr := opt.AfterInitialize(db); errr != nil {
```

```
16                 err = errr
17             }
18         }(opt)
19     }
20 }
```

可以看到，对每一个 option，我们直接调用它的 Apply 方法来对数据库的配置 config 进行修改。Option 的这种编程方式常用在初始化一个比较复杂的结构里面。

比如这里在 Gorm 中，要初始化一个 Gorm 的构造配置 gorm.Config，而这个 Config 结构有非常多的配置项，我们希望在创建初始化的时候，能对这个配置进行调整。所以就可以在 Option 方法中再定义一个 Apply 方法，它的参数是 gorm.Config 指针：

复制代码

```
1 func (c *Config) Apply(config *Config) error
```

这样，可以遍历所有的 Option，挨个调用它们的 Apply 方法对 Config 进行设置，最终我们获取的就是经过所有 Option 处理后的 Config。

这种 Option 的编程方法在 Golang 中十分常用，要好好掌握，下一节课，我们也会用这种方式为 hade 的 ORM 服务来注册参数。

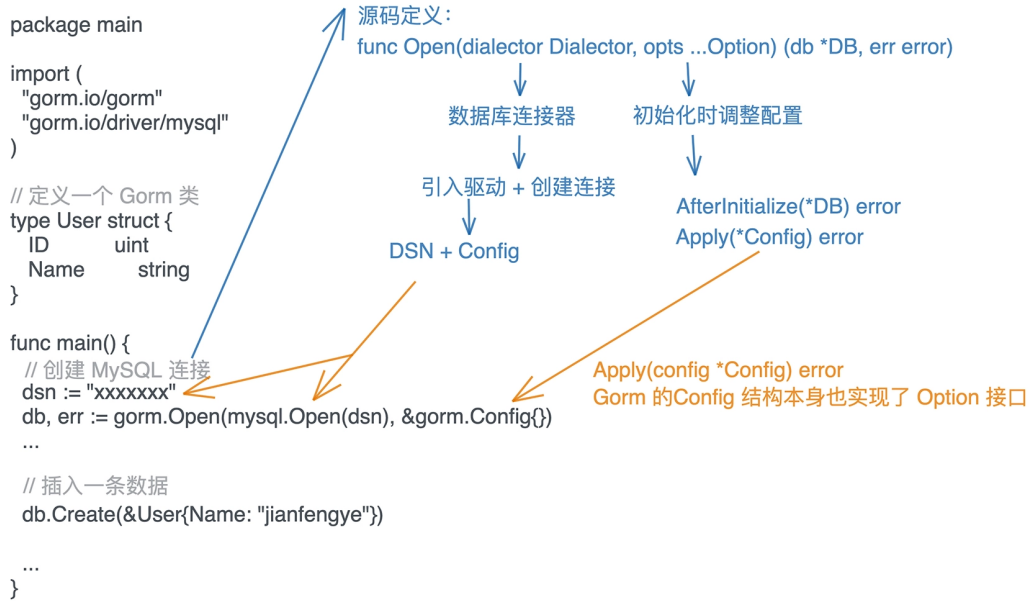
Gorm 这里还有一个比较巧妙的设计，Config 结构本身也实现了 Option 接口。按照这个设计实现之后，你会发现，Config 本身也可以作为一个 Option 在 Open 的第二个参数中出现。

复制代码

```
1  func (c *Config) Apply(config *Config) error {
2      if config != c {
3          *config = *c
4      }
5      return nil
6  }
7
8  func (c *Config) AfterInitialize(db *DB) error {
9      if db != nil {
10         for _, plugin := range c.Plugins {
11             if err := plugin.Initialize(db); err != nil {
12                 return err
```

```
            }
        }
    }
    return nil
}
```

讲到这里相信你能画出第二个参数的要点了，Gorm 实现的时候使用的是 gorm.Config，之所以它可以匹配 Open 函数定义的 Option 的参数的原因是 Config 结构本身也实现了 Option 接口。



通过上图我们就理解了 Open 函数在使用的时候，第一个参数和第二个参数是如何对应函数定义两个参数的了。

所以 Gorm 官方连接 MySQL 的示例就很好理解了，看注释：

复制代码

```
import (
  "gorm.io/driver/mysql"
  "gorm.io/gorm"
)
func main() {
  // 参考 https://github.com/go-sql-driver/mysql#dsn-data-source-name 获取详情
  dsn := "user:pass@tcp(127.0.0.1:3306)/dbname?charset=utf8mb4&parseTime=True&
  // 第一个参数是 dialector
  // 第二个参数是 option，但是由于 gorm.Config 实现了 option，所以可以这么使用
```

```
10    db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
11 }
```

gorm.DB

两个传入参数讲完了，我们继续看 Open 的返回结构，除了常规的 error 外，还有一个 gorm.DB 的结构指针，定义如下：

复制代码

```
1 type DB struct {
2   *Config
3   Error        error
4   RowsAffected int64
5   Statement    *Statement
6   // Has unexported fields.
7 }
```

它具有丰富的操作数据库的方法，比如增加数据的 Create 方法、更新数据的 Update 方法。

我们研究下 gorm.DB 的结构，它嵌套了一层 gorm.Config 结构，里面有几个关键字段：

复制代码

```
1  // Config GORM config
2  type Config struct {
3     ...
4     // gorm 的日志输出
5     Logger logger.Interface
6     ...
7
8     // db 的具体连接
9     ConnPool ConnPool
10    // db 驱动器
11    Dialector
12    ...
13
14    callbacks  *callbacks  // 回调方法
15    ...
16 }
```

其中的 Logger 、 ConnPool 和 Callback 字段，值得详细研究一下。

Logger

我们从 Open 看到了，一个 gorm.DB 结构就代表一个数据库连接，而这个数据库连接的所有日志操作输出在哪里呢？就是通过这个 Logger 字段配置的。

Logger 字段是一个接口，表示如果有一个实现了 logger.Interface 接口的日志输出类，我就能让这个 DB 的所有数据库操作的日志，都输出到这个类中。

复制代码

```
// Interface logger interface
type Interface interface {
    LogMode(LogLevel) Interface
    Info(context.Context, string, ...interface{})
    Warn(context.Context, string, ...interface{})
    Error(context.Context, string, ...interface{})
    Trace(ctx context.Context, begin time.Time, fc func() (sql string, rowsAffe
}
```

Gorm 使用 Logger 接口的方法，和我们 hade 框架定义 Logger 服务的方法如出一辙，它**不定义具体的实现类，而是定义了具体的接口**。所以下一节课，我们将 Gorm 融合进入 hade 框架的时候，要做的事情就是封装一个实现了 Gorm 的 logger.Interface 接口的实现类，而这个实现类的具体实现方法，使用 hade 框架的日志服务类来实现。

ConnPool

ConnPool 也定义了一个接口，它代表数据库的真实连接所在的连接池。这个接口的定义，我认为是 Gorm 中最精妙的一个地方了：

复制代码

```
// ConnPool db conns pool interface
type ConnPool interface {
    PrepareContext(ctx context.Context, query string) (*sql.Stmt, error)
    ExecContext(ctx context.Context, query string, args ...interface{}) (sql.Re
    QueryContext(ctx context.Context, query string, args ...interface{}) (*sql.
    QueryRowContext(ctx context.Context, query string, args ...interface{}) *sq
}
```

这个接口定义了四个方法，但它们并不是随便定义的，而是根据 Golang 标准库的 database/sql 的 Conn 结构来定义的，这是什么意思呢？

首先我们要知道，Golang 的标准库 database/sql 其实定义了一套数据库连接规范。官方的基本思想就是，数据库的种类非常多，我不可能对每一个数据库都实现一套定制化的类库，所以我定义一套基本数据结构和方法，并且提供每个数据库需要实现的驱动接口。**使用者只需要实现驱动接口，就能使用这套基本数据结构和方法了。**

是不是和前面说的 Gorm 的驱动逻辑一样？是的。Golang 中所有的 ORM 库，底层都是基于标准库的 database/sql 来实现数据库的连接和基本操作。只是在具体操作上，会封装一层逻辑，当使用不同驱动接口的时候，实现不一样的接口操作。

这里的 ConnPool 就是 Gorm 对 database/sql 的数据结构的封装。换句话说，开头的 Gorm 使用例子，在底层 database/sql 的简要实现大致如下：

复制代码

```
package main

import (
  "database/sql"
)

func main() {

  dsn := "xxxx"
  ...
  db, err = sql.Open("mysql", *dsn)
  ...

    result, err := db.ExecContext(ctx, "INSERT INTO user (name) values ('jianf
    ...
}
```

这里 sql.Open 创建的 sql.DB 结构，就包含 ConnPool 中定义的四个接口：PrepareContext、ExecContext、QueryContext、QueryRowContext。也就是说：**database/sql 的 sql.DB 结构实现了 Gorm 库的 ConnPool 接口。**

而实际上，database/sql 里面的 sql.DB 结构就是一个连接池结构，我们可以通过以下四个方法设置连接池的不同属性：

复制代码

```
// 设置连接的最大空闲时长
```

```
func (db *DB) SetConnMaxIdleTime(d time.Duration)
// 设置连接的最大生命时长
func (db *DB) SetConnMaxLifetime(d time.Duration)
// 设置最大空闲连接数
func (db *DB) SetMaxIdleConns(n int)
// 设置最大打开连接数
func (db *DB) SetMaxOpenConns(n int)
```

所以 gorm.DB 里面的 ConnPool 实际上存放的就是 database/sql 的 sql.DB 结构。

callbacks

最后看 gorm.DB 里面的 callbacks 字段，它存放的是所有具体函数的调用方法。callback 指针指向的数据结构也是叫做同名的 callbacks：

复制代码

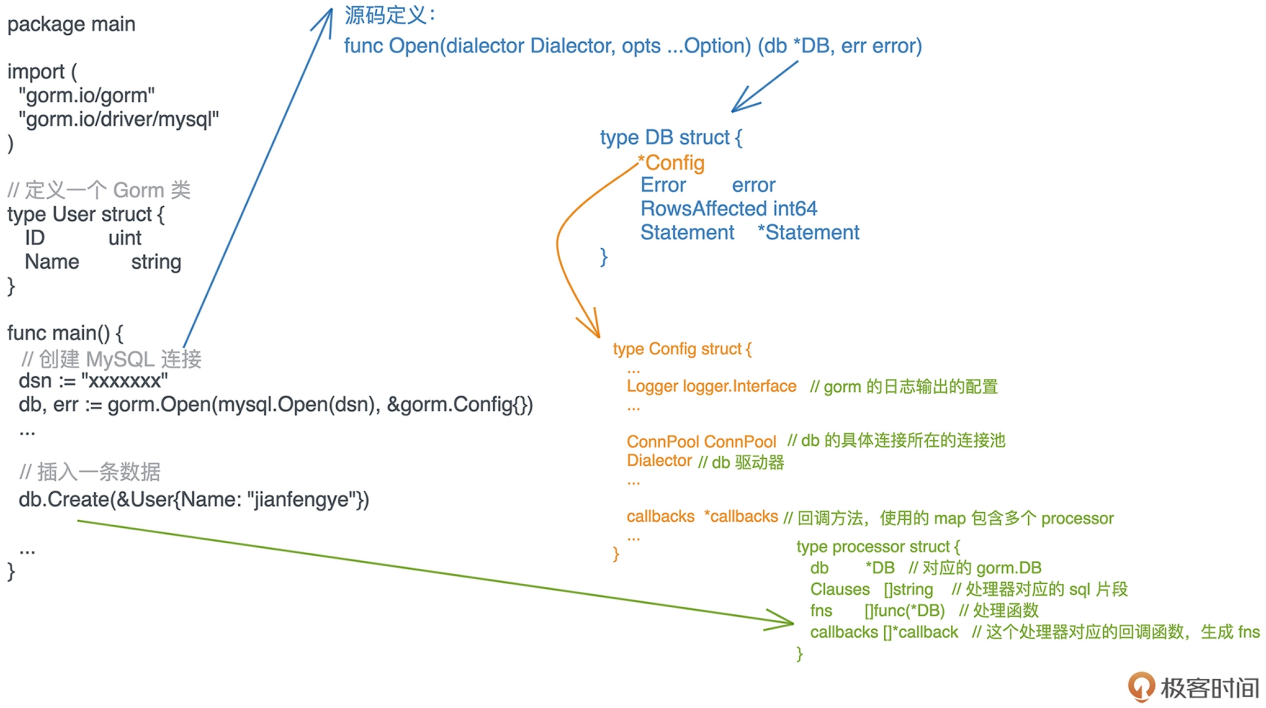
```
// callbacks gorm callbacks manager
type callbacks struct {
    processors map[string]*processor
}
```

它里面使用的 map 包含多个 processor。一个 processor 就是一种操作的处理器。processer 的结构定义为：

复制代码

```
type processor struct {
    db        *DB // 对应的 gorm.DB
    Clauses   []string  // 处理器对应的 sql 片段
    fns       []func(*DB) // 这个处理器对应的处理函数
    callbacks []*callback // 这个处理器对应的回调函数，生成 fns
}
```

开头的那个例子，我们调用了 gorm.DB 的 Create 方法，它会去 gorm.DB 的 callbacks 中的 processors 里，寻找 key 为“create”的处理器 processor。然后逐个调用处理器中设置好的 fns。下面分析源码的时候也会看到具体的实现逻辑。



源码

现在理解了 Gorm 在创建连接过程中涉及的几个关键对象，我们就再从源码开始梳理一下 Gorm 的核心逻辑，理解下 Gorm 是怎么使用 Open 创建数据库连接、怎么使用创建的数据库连接的 Create 方法来创建一条数据的。再把开头官网的例子拿出来。

复制代码

```
package main

import (
  "gorm.io/gorm"
  "gorm.io/driver/mysql"
)

type Product struct {
  gorm.Model
  Code  string
  Price uint
}

func main() {
  dsn := "xxxxxxx"
  db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
  if err != nil {
    panic("failed to connect database")
  }

  ...
```

```
22
23      // Create
24      db.Create(&Product{Code: "D42", Price: 100})
25
26      ...
27  }
```

用第一节课教的思维导图的方式来分析这个 Gorm 的主流程，主要就是 gorm.Open 和 db.Create 两个方法。

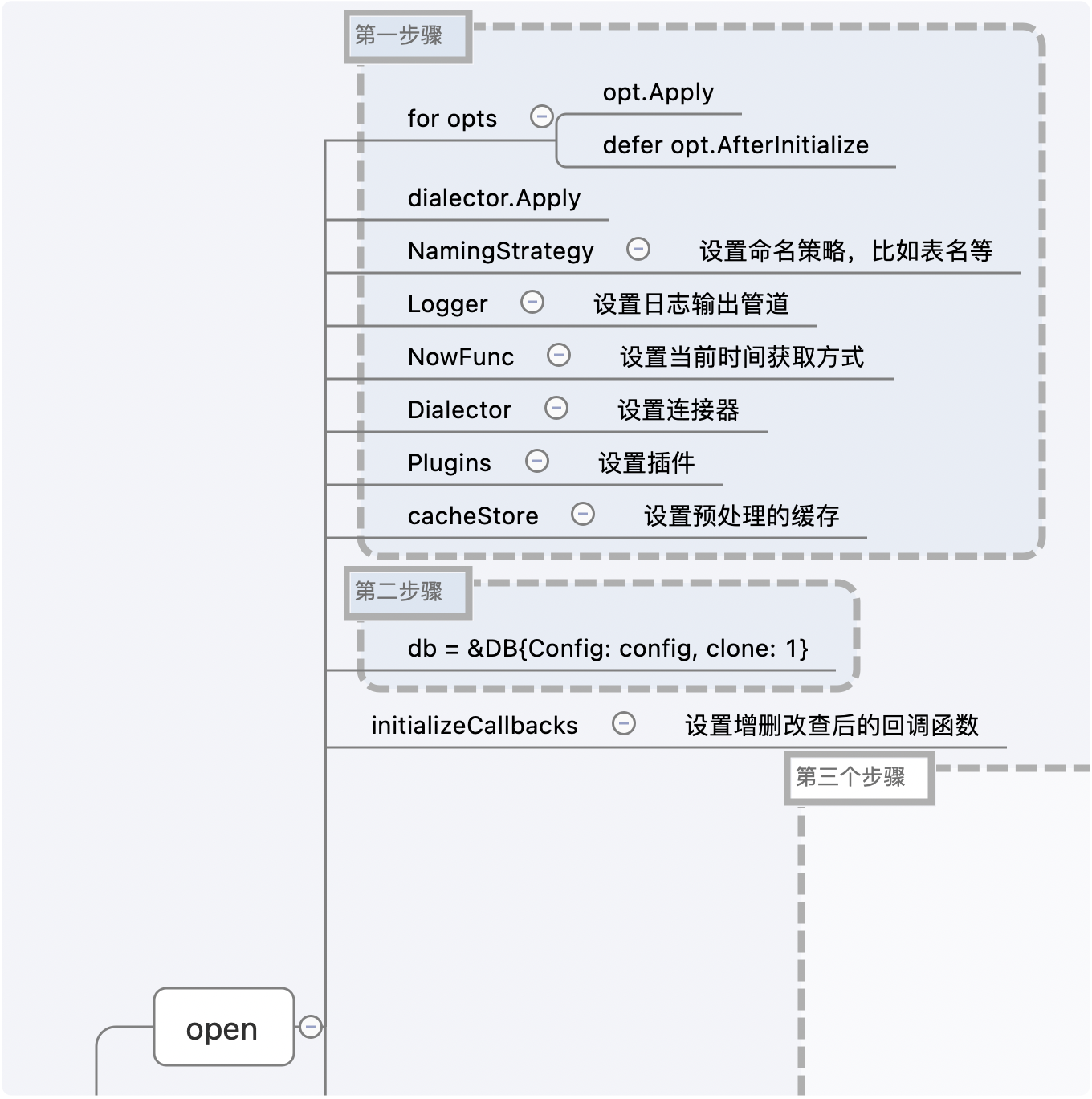
gorm.Open

首先是 gorm.Open：

复制代码

```
1   db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
```

我们将函数源码分为四个大步：



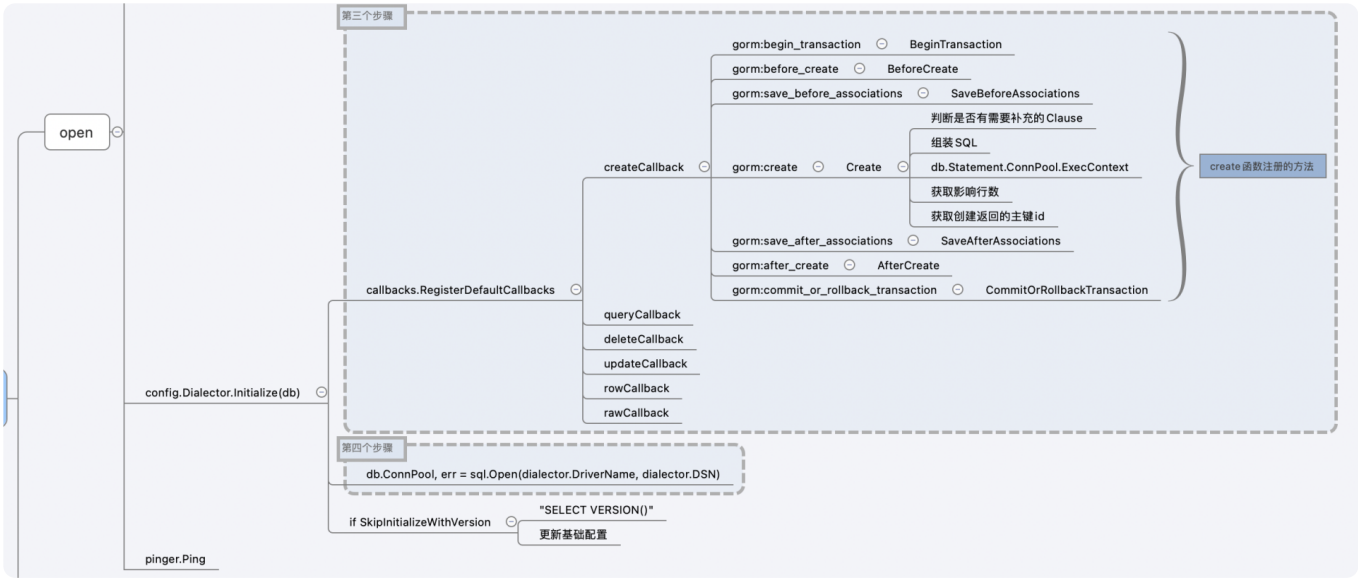
第一大步，初始化 gorm.Config 结构。通过使用参数中的 Option 可变参数的 Apply 接口，对最终的配置结构 gorm.Config 进行相应的修改，其中包括修改输出的 Logger 结构。

第二步，初始化 gorm.DB 结构：

```
1  db = &DB{Config: config, clone: 1}
```

复制代码

第三步，初始化 gorm.DB 的 callbacks。



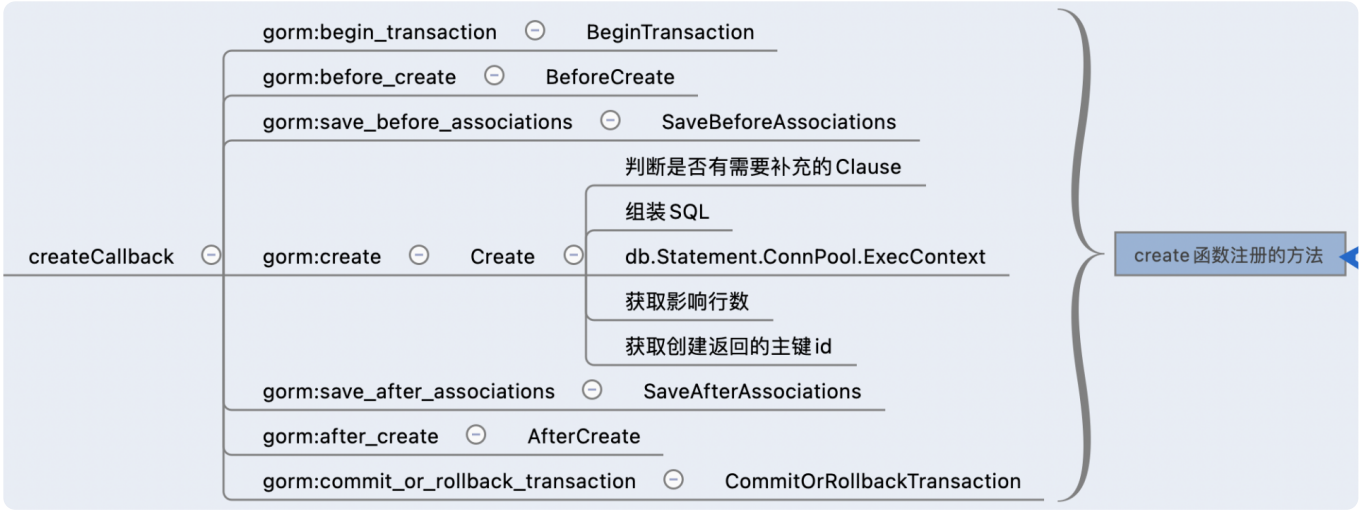
这里我们只拆解了这个例子的 create 函数相关的 callback。核心的关键函数在 Gorm 库 callback.go 的 RegisterDefaultCallbacks 方法。比如下列的代码，就是创建 create 相关的执行方法 fns：

复制代码

```
createCallback := db.Callback().Create()
createCallback.Match(enableTransaction).Register("gorm:begin_transaction", Beg
createCallback.Register("gorm:before_create", BeforeCreate)
createCallback.Register("gorm:save_before_associations", SaveBeforeAssociation
createCallback.Register("gorm:create", Create(config))
createCallback.Register("gorm:save_after_associations", SaveAfterAssociations(
createCallback.Register("gorm:after_create", AfterCreate)
createCallback.Match(enableTransaction).Register("gorm:commit_or_rollback_tran
if len(config.CreateClauses) == 0 {
    config.CreateClauses = createClauses
}
createCallback.Clauses = config.CreateClauses
```

我们可以看到，Gorm 在一个 create 方法，定义了 7 个执行方法 fns，分别是：BeginTransaction、BeforeCreate、SaveBeforeAssociations、Create、SaveAfterAssociations、AfterCreate、CommitOrRollbackTransaction。这七个执行方法就是按照顺序，从上到下每个 Create 函数都会执行的方法。

其中关注一下 Create 方法，它又分为五个步骤：



我们看到了熟悉的 ExecContent 函数，这个就对应上了 Golang 标准库的 database/sql 中 sql.DB 的 ExecContext 方法。原来它藏在这里！

那前面说的 database/sql 的 sql.DB 的 Open 方法，又放在哪里呢？就在 gorm.Open 的第四大步中：

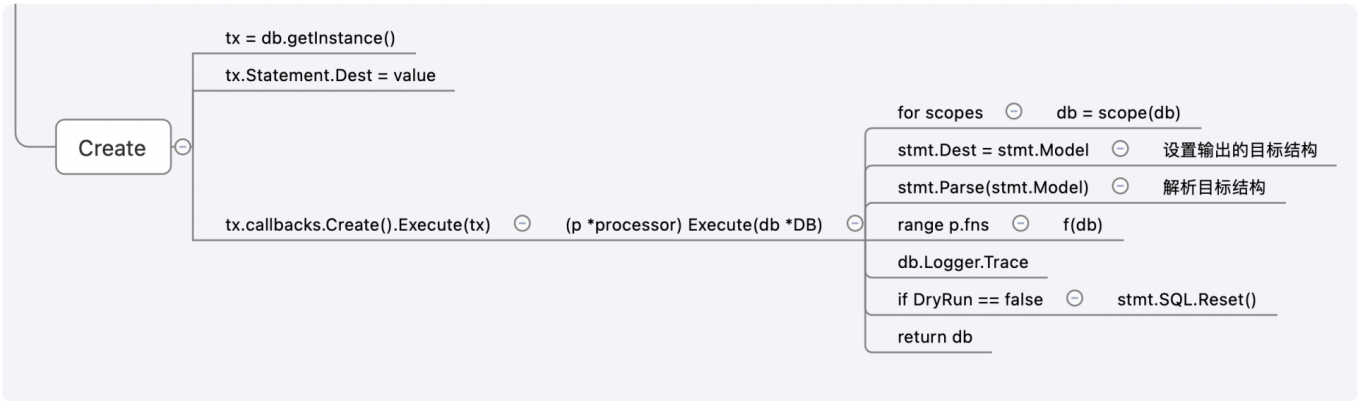
```
1 db.ConnPool, err = sql.Open(dialector.DriverName, dialector.DSN)
```

复制代码

将 database/sql 中生成的 sql.DB 结构，设置在了 gorm.DB 的 ConnPool 上。

db.Create

下面再来看 gorm.DB 的 Create 方法。它的任务就很简单了：触发启动 processor 中的 fns 方法。具体最核心的代码就在 Gorm 的 callback.go 中的 Execute 函数里。



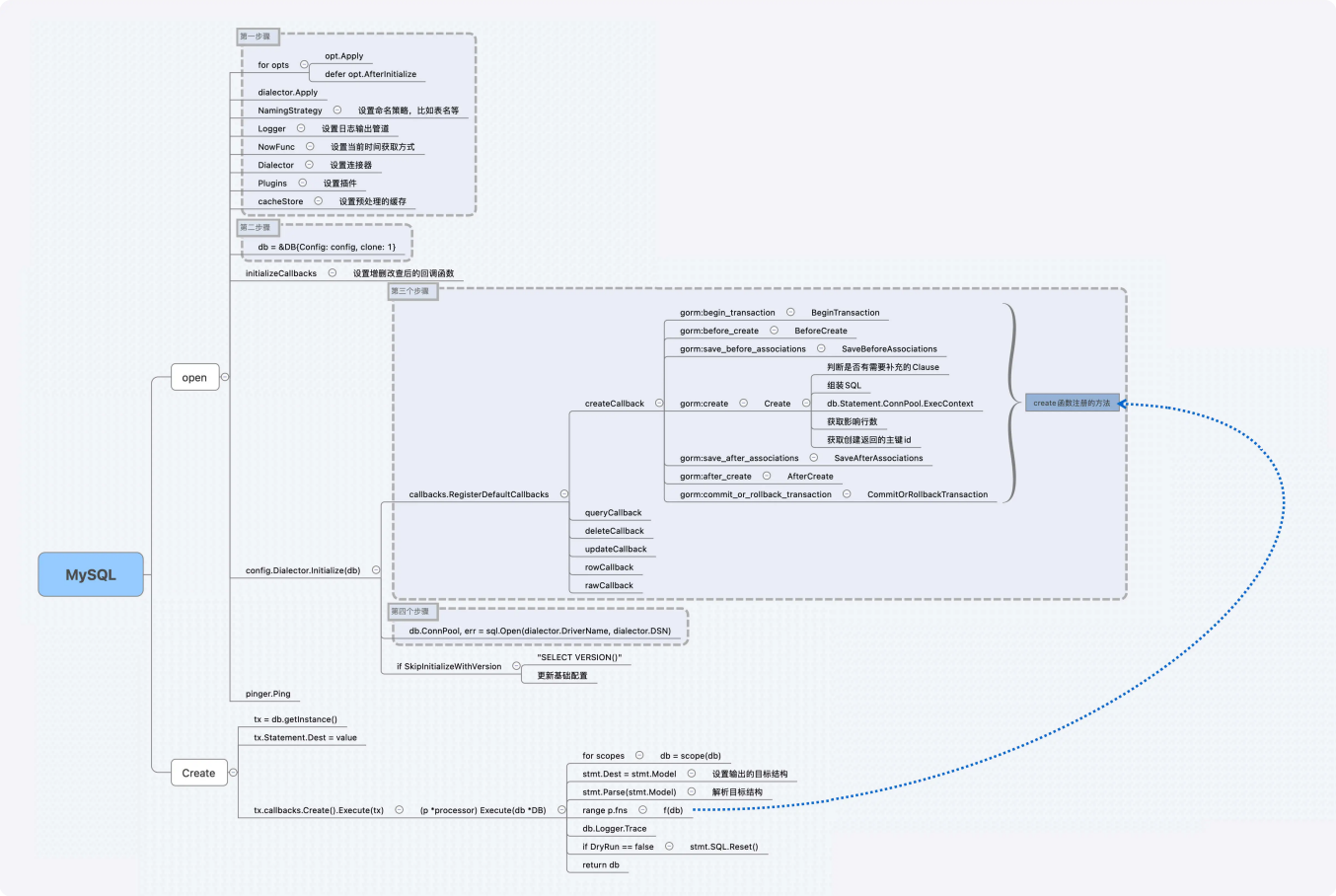
可以看到，在 Execute 函数中，最核心的是遍历 fns，调用 fn(db) 方法，其中就有我们前面定义的 Create 方法了，也就是执行了 database/sql 的 db.ExecContext 方法。

这里我们就根据思维导图找到了 Gorm 封装的 database/sql 的两个关键步骤：

sql.Open

db.ExecContext

理解了这一点，就基本理解了 Gorm 最核心的实现原理了。



当然 Gorm 中还有一个部分，是将我们定义的 Model 解析成为 SQL 语句，这里又是 Gorm 定义的一套非常庞大的数据结构支撑的了，其中包括 Statement、Schema、Field、Relationship 等和数据表操作相关的数据结构。

这需要用另外一个篇幅来描述了。不过这块 Model 解析，对我们下一章 hade 框架融合 Gorm 的影响并不大。有兴趣的同学可以追着上述 Create 方法中的 stmt.Parse 方法进一步分析。

今天我们还没有涉及代码修改，思维导图保存在 GitHub 上的 geekbang/25 分支中根目录的 mysql.xmind 中了。

小结

我们分析了 Gorm 的具体数据结构和创建连接的核心源码流程。想要检验自己是否理解这节课也很简单，你可以对照开头为 user 表插入一行的代码，看看能不能清晰分析出它的底层是如何封装标准库的 database/sql 来实现的。

我们在阅读 Gorm 源码的同时，也是在学习它的优秀编码方式，比如今天讲到的 Option 方式、定义驱动、ConnPool 定义实现标准库方法的接口。这些都是 Gorm 设计精妙的地方。

当然 Gorm 的代码远不是一篇文章能说透的。其中包含的 Model 解析，以及更多的具体细节实现，都得靠你在后续使用过程中多看官网、多思考、多解析，才能完全吃透这个库。

思考题

GORM 有一个功能我非常喜欢，DryRun 空跑，这个设置是在 gorm.DB 结构中的。如果我们设置了 gorm.DB 的 DryRun，能让我在这个 DB 中的所有 SQL 操作并不真正执行，这个功能在调试的时候是非常有用的。你能再顺着思维导图，分析出 DryRun 是怎么做到这一点的么？

欢迎在留言区分享你的思考。感谢你的收听，如果你觉得今天的内容对你有所帮助，也欢迎分享给身边的朋友，邀请他一起学习。我们下节课见～

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 24 | 管理进程：如何设计完善的运行命令？

下一篇 26 | GORM：数据库的使用必不可少（下）

精选留言（1）

写留言

Vincent 神

2021-11-15

对优秀的开源代码，应该吃透

展开